

3.1 Neural Networks and Backpropagation

CS 6301
Spring 2023

Outline - Key Concepts

NLP

N/A

ML

Neural Networks

Forward Propagation

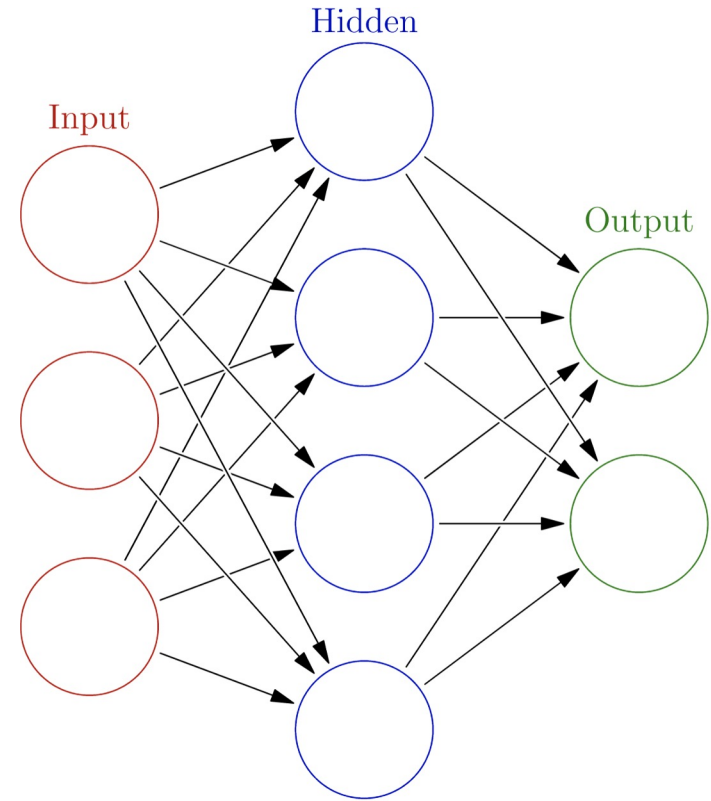
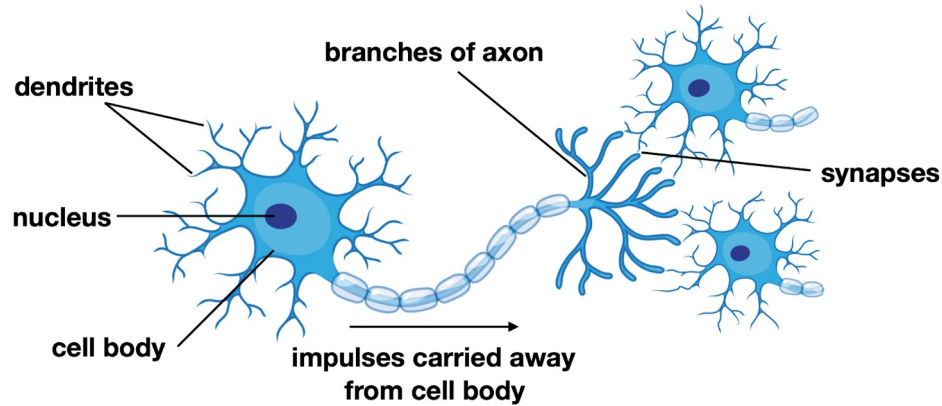
Gradient Descent

Backpropagation

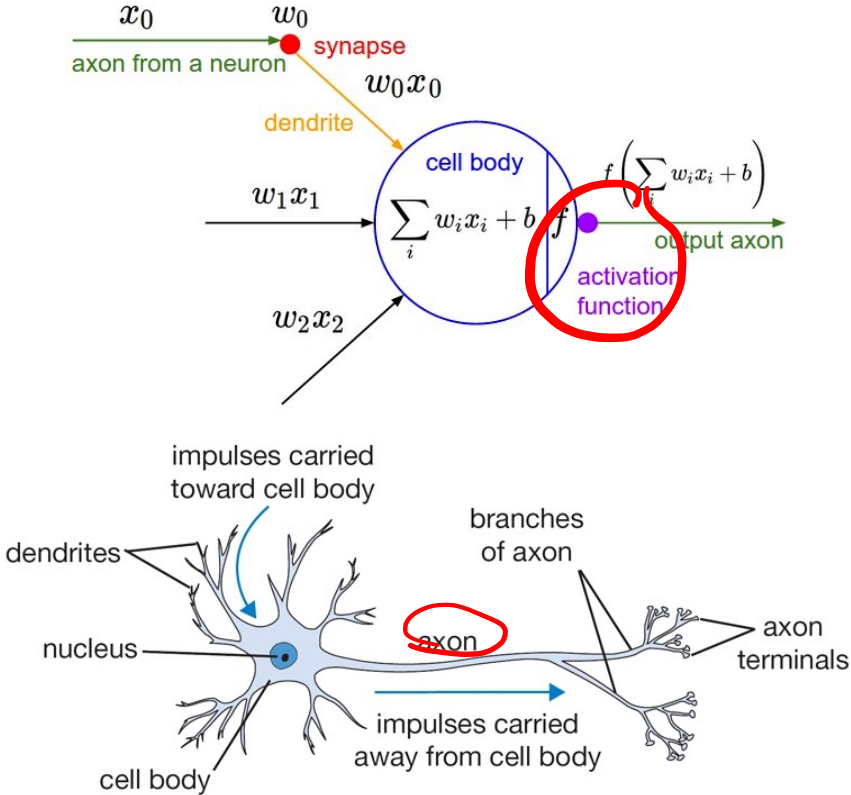
Computational Graph

Optimizers

Neural Networks == Deep Learning



Biological “Inspiration”



Some History

- Neural network algorithms date to the 80's
 - Originally inspired by early neuroscience
- Historically slow, complex, and unwieldy
- Now: term is abstract enough to encompass almost any model – but useful!
- Dramatic shift in last 3-4 years away from MaxEnt/LR (**linear, convex**) to “neural net” (non-linear architecture, **non-convex**)

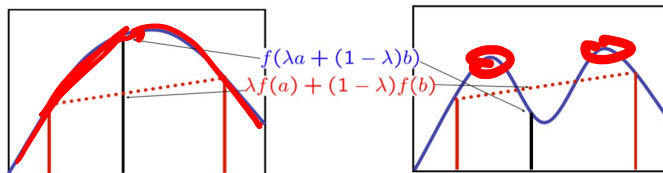
Linearity, Convexity

- MaxEnt (linear, convex) to “neural net” (non-linear architecture, non-convex)

The MaxEnt objective behaves nicely:

- Differentiable (so many ways to optimize)
- Convex (so no local optima)

$$f(\lambda a + (1 - \lambda)b) \geq \lambda f(a) + (1 - \lambda)f(b)$$



Convex

LR

Non-Convex

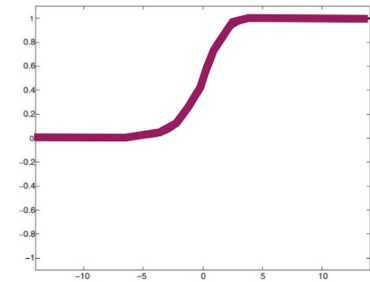
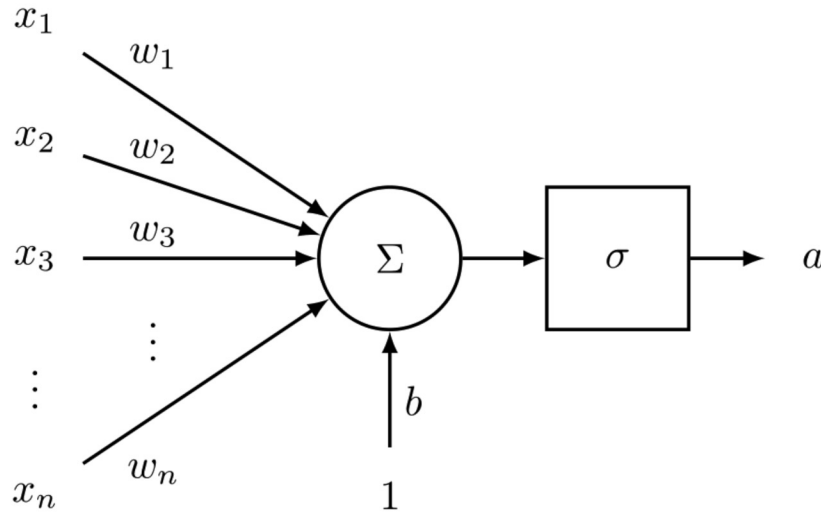
Convexity guarantees a single, global maximum value because any higher points are greedily reachable

sci learn

(Starting from) A Neuron



A neuron is the fundamental building block of neural networks.
e.g., a neuron with sigmoid function



Sigmoid

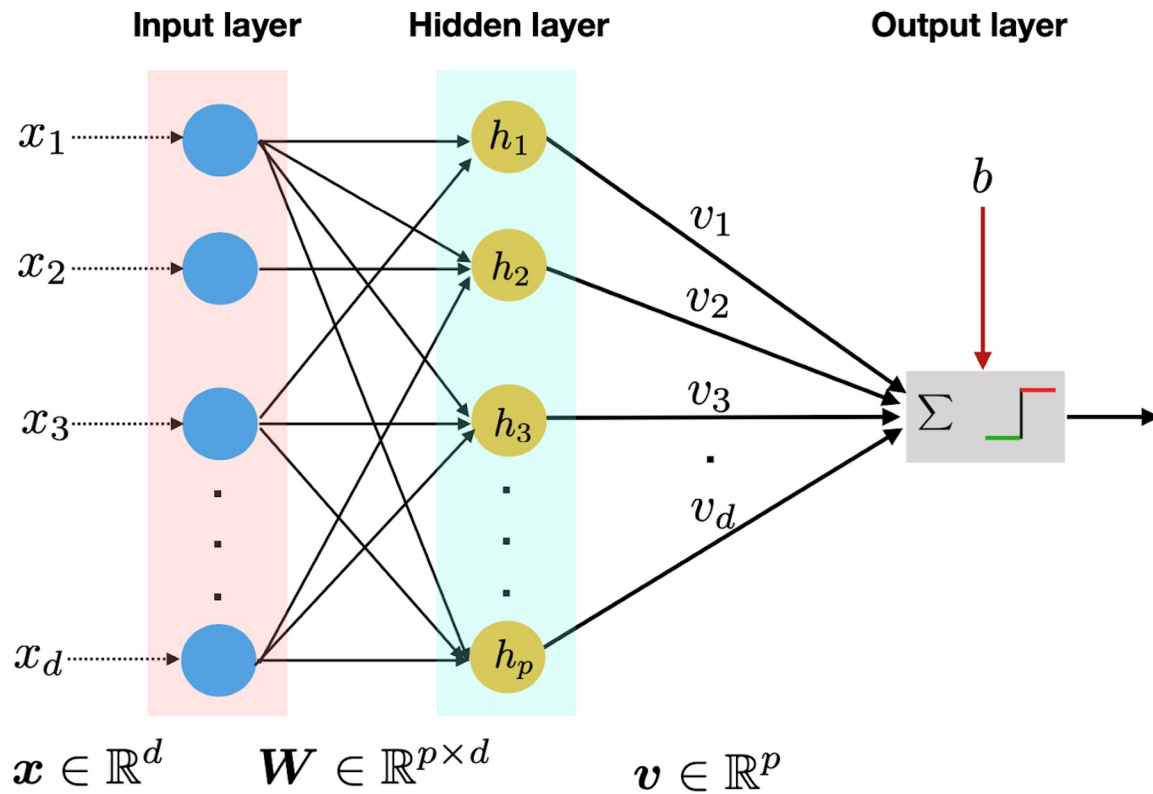
$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

$$a = \frac{1}{1 + \exp(-(w^T x + b))}$$

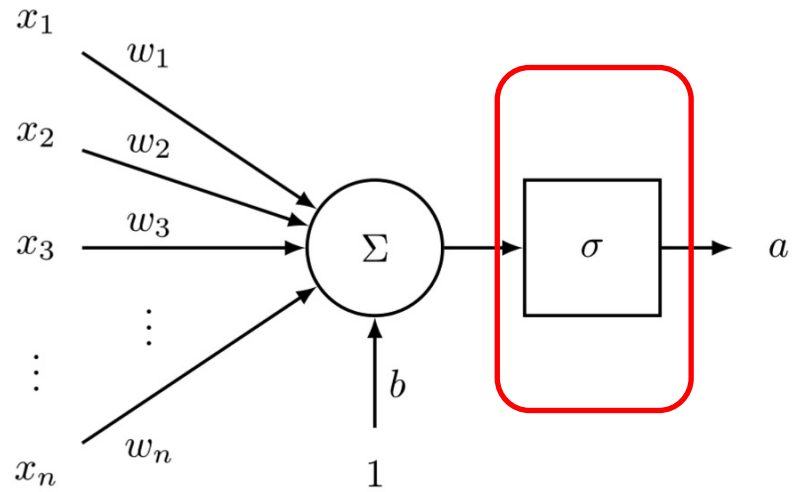
Presentations

Projects

Two layer neural networks

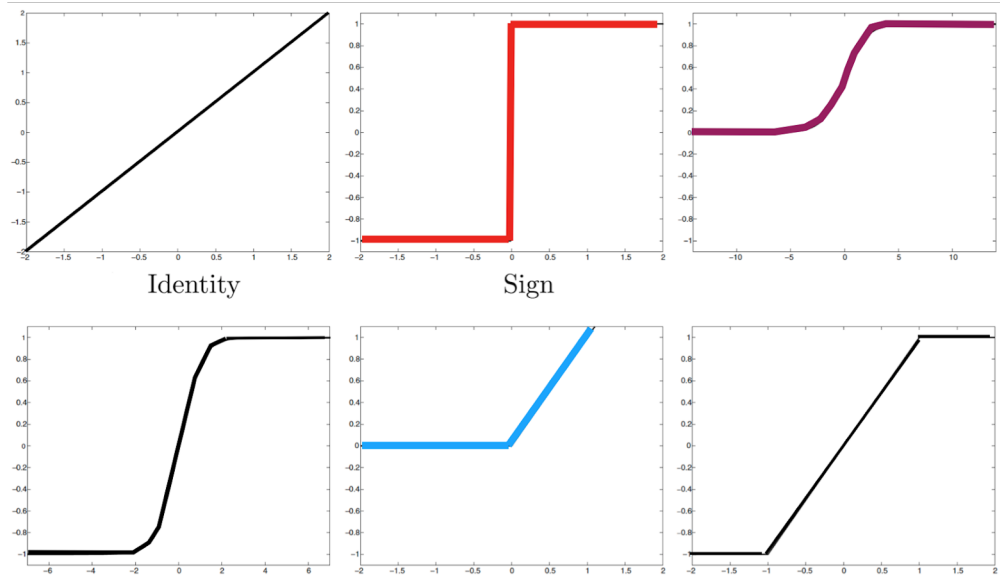


Activation Functions



$$a = \frac{1}{1 + \exp(-(w^T x + b))}$$

Activation Functions



Identity

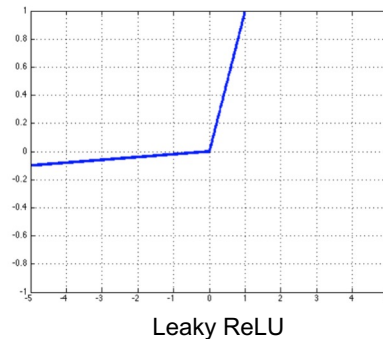
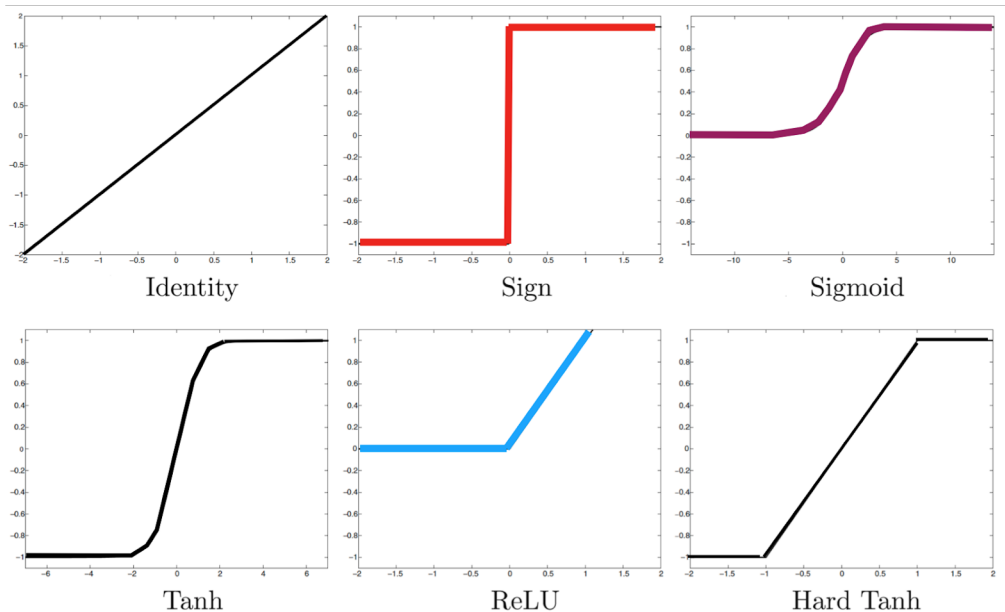
Sign

$$\sigma(z) = \text{sign}(z) \quad (\text{Sign})$$

$$\sigma(z) = \max\{0, z\} \quad (\text{Rectified Linear Unit [ReLU]})$$

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (\text{Sigmoid})$$

Activation Functions



$$\text{leaky}(z) = \max(z, k \cdot z)$$

where $0 < k < 1$

$$\sigma(z) = \text{sign}(z) \quad (\text{Sign})$$

$$\sigma(z) = \max\{0, z\} \quad (\text{Rectified Linear Unit [ReLU]})$$

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (\text{Sigmoid})$$

$$\tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

$$\text{hardtanh}(z) = \begin{cases} -1 & : z < -1 \\ z & : -1 \leq z \leq 1 \\ 1 & : z > 1 \end{cases}$$

How to train Neural Networks?

- Start from two-layer perceptron.

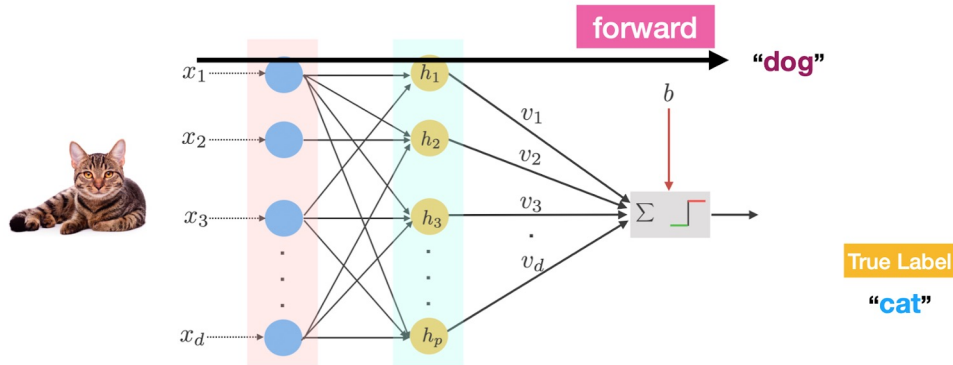
How to train two-layer Perceptron?

Forward propagation:

Backward propagation:

How to train two-layer Perceptron?

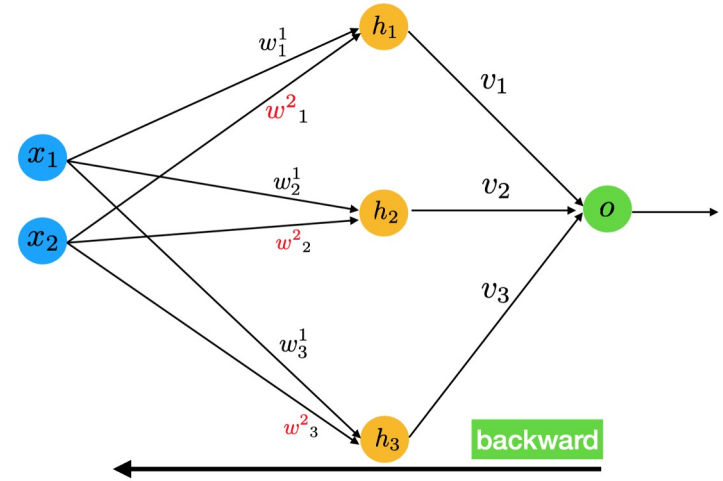
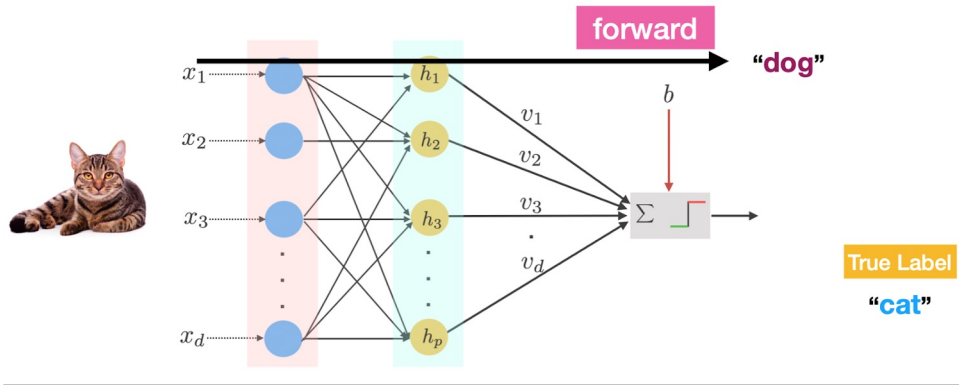
Forward propagation: In this phase, the inputs for a training instance are fed into the neural network. This results in a forward cascade of computations across the layers, using the current set of weights.



Backward propagation:

How to train two-layer Perceptron?

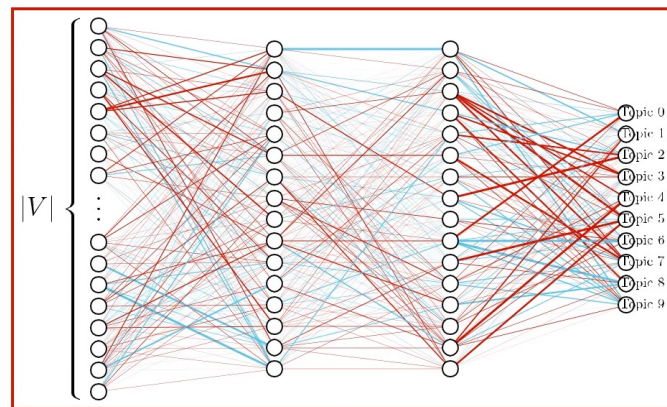
Forward propagation: In this phase, the inputs for a training instance are fed into the neural network. This results in a forward cascade of computations across the layers, using the current set of weights. The output is the prediction of current weights.



Backward propagation: The main goal of the backward phase is to **learn the gradient of the loss function** with respect to the different weights by **?**.

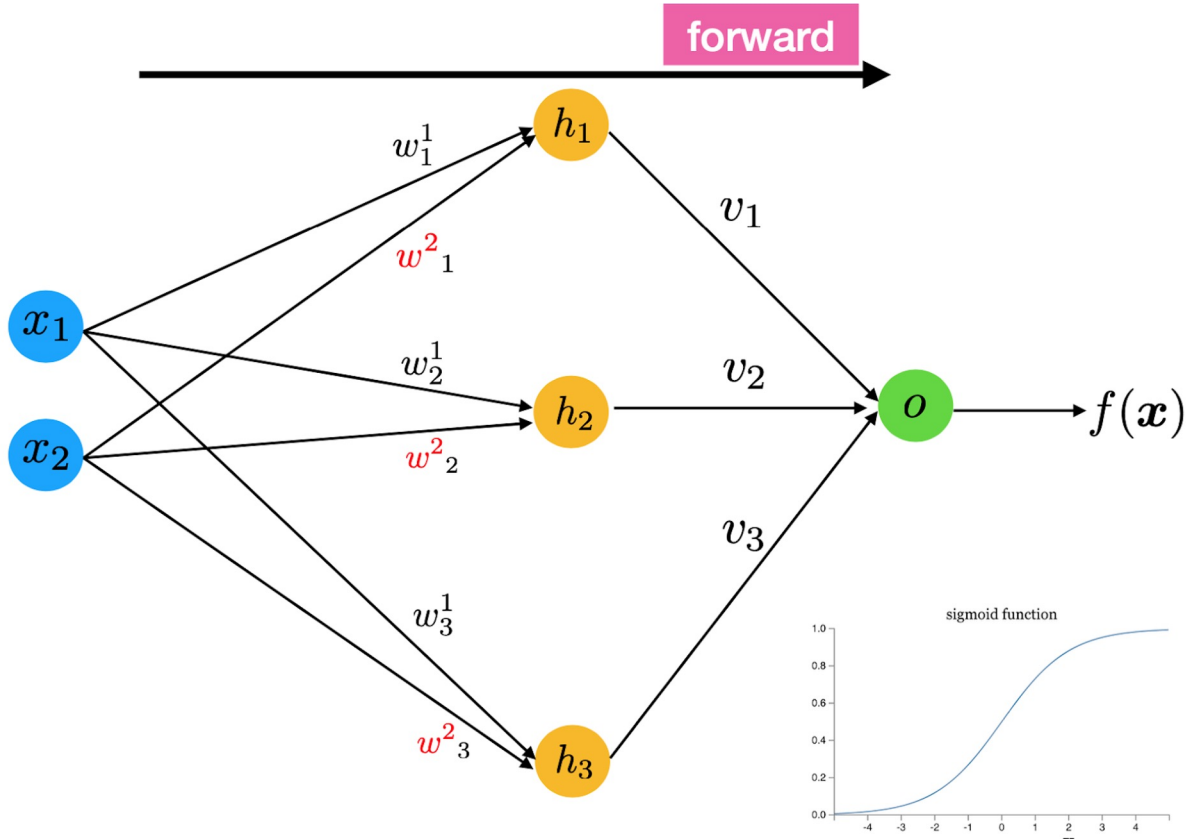
Animation

$$-\nabla C(\underbrace{\dots}_{\text{All weights and biases}}) = \begin{bmatrix} 0.20 \\ 0.83 \\ -0.84 \\ \vdots \\ 0.04 \\ 1.57 \\ 1.59 \end{bmatrix}$$



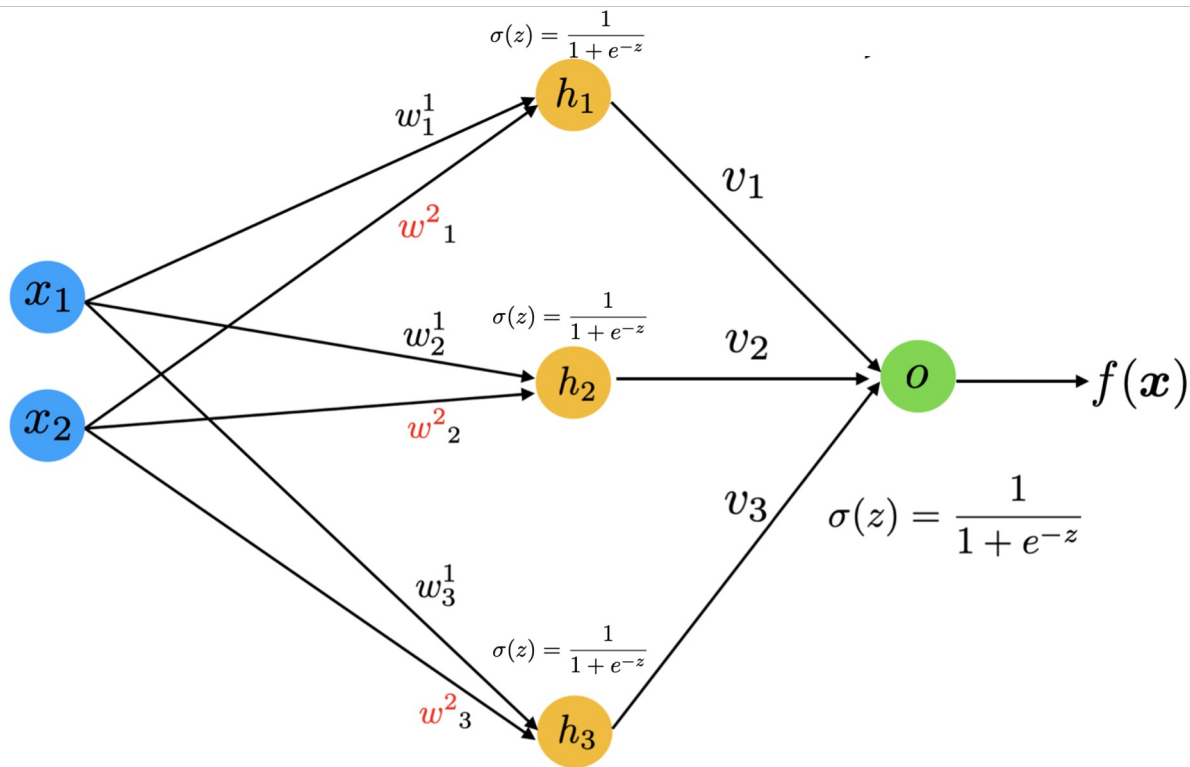
$$C(w_0, w_1, \dots, w_{13,001}) = 3.4$$

Forward propagation

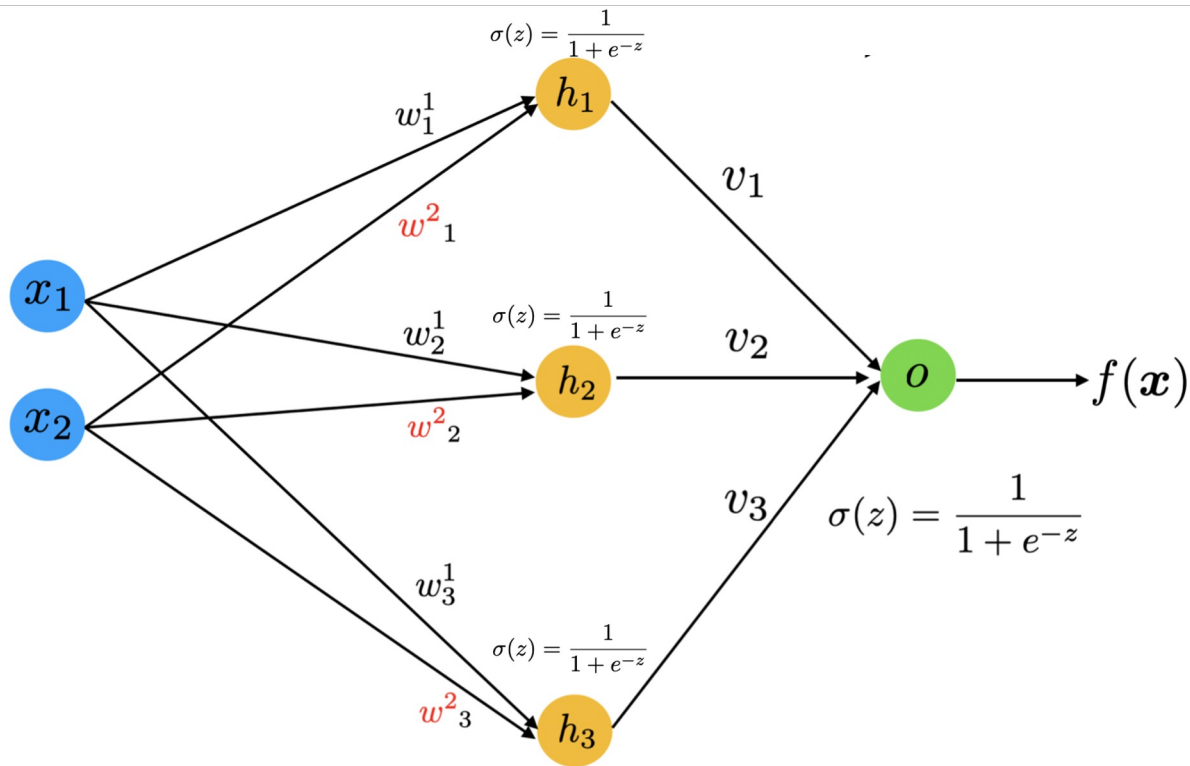


We use the **sigmoid** activation function which introduces better non-linearity:

Forward propagation



Forward propagation



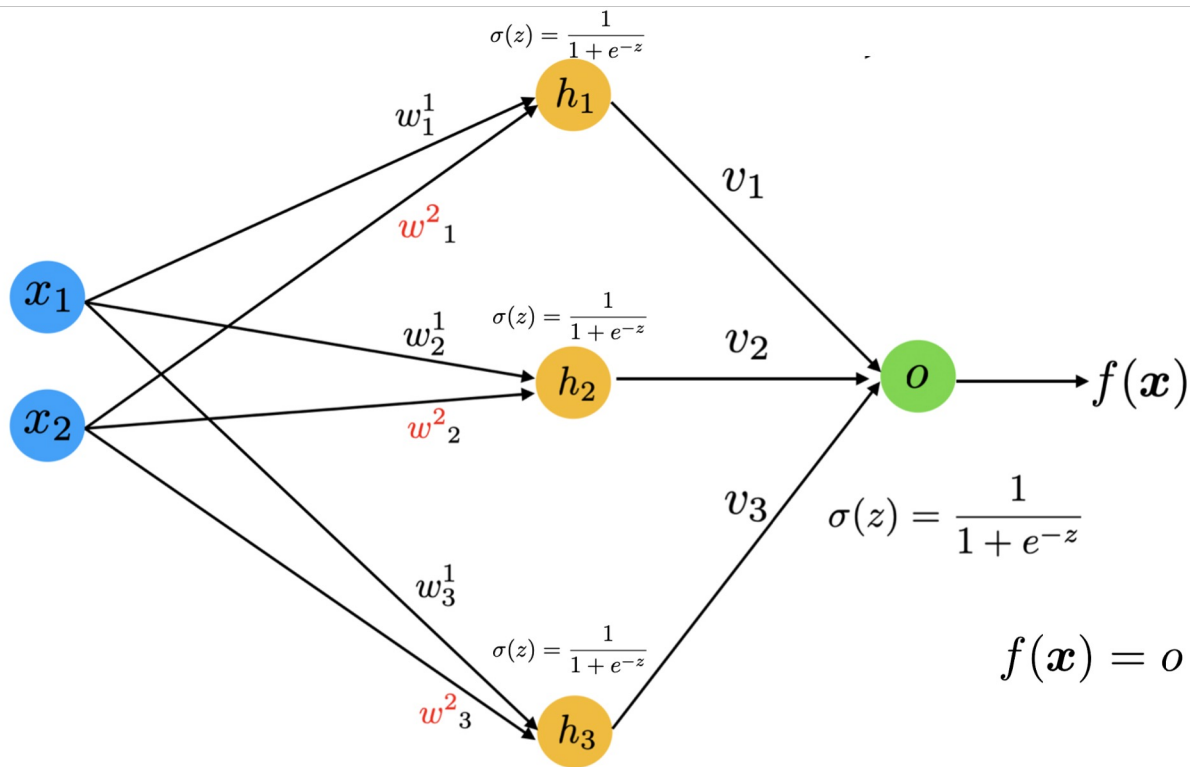
$$h_1 = \sigma(w_1^1 x_1 + w_1^2 x_2)$$

$$h_2 = \sigma(w_2^1 x_1 + w_2^2 x_2)$$

$$h_3 = \sigma(w_3^1 x_1 + w_3^2 x_2)$$

$$o = \sigma(v_1 h_1 + v_2 h_2 + v_3 h_3)$$

Forward propagation



$$h_1 = \sigma(w_1^1 x_1 + w_1^2 x_2)$$

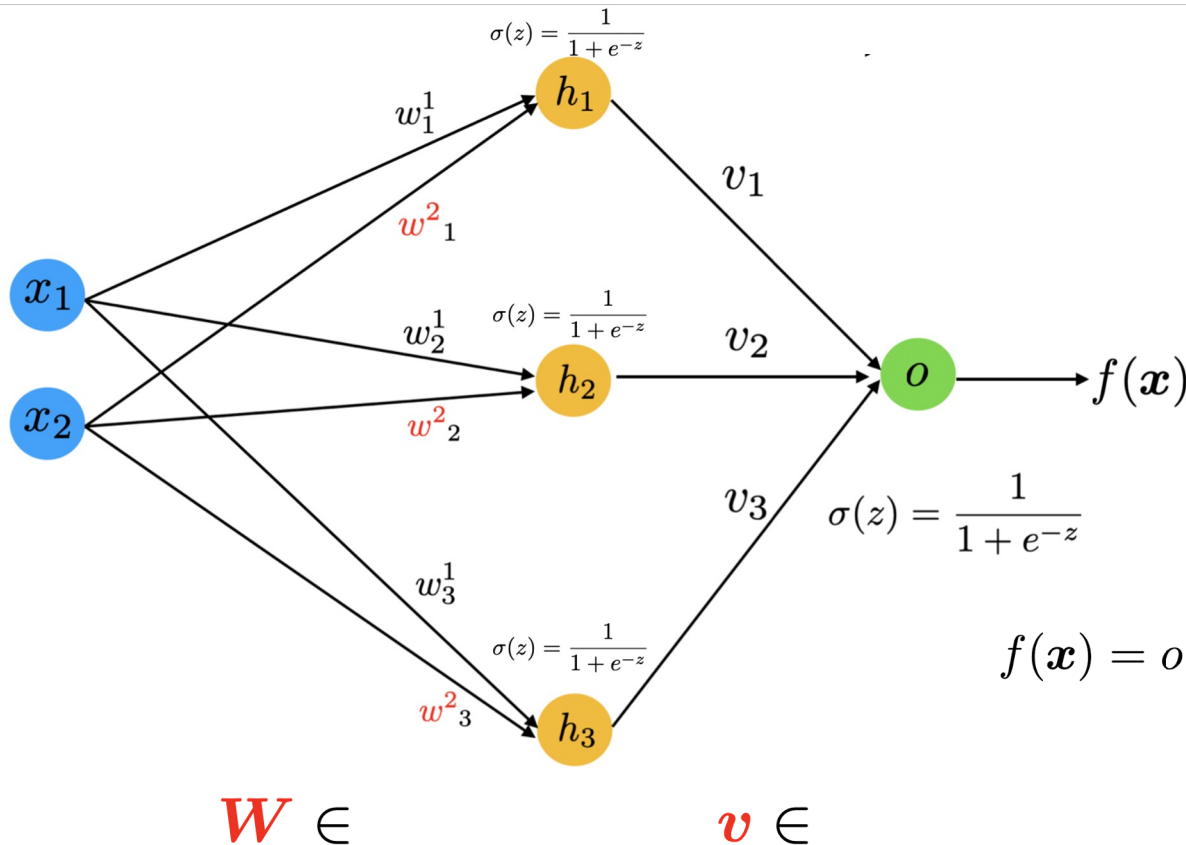
$$h_2 = \sigma(w_2^1 x_1 + w_2^2 x_2)$$

$$h_3 = \sigma(w_3^1 x_1 + w_3^2 x_2)$$

$$o = \sigma(v_1 h_1 + v_2 h_2 + v_3 h_3)$$

$$f(\mathbf{x}) = o = \sigma(v_1 h_1 + v_2 h_2 + v_3 h_3)$$

Forward propagation - Compact Form



$$h_1 = \sigma(w_1^1 x_1 + w_1^2 x_2)$$

$$h_2 = \sigma(w_2^1 x_1 + w_2^2 x_2)$$

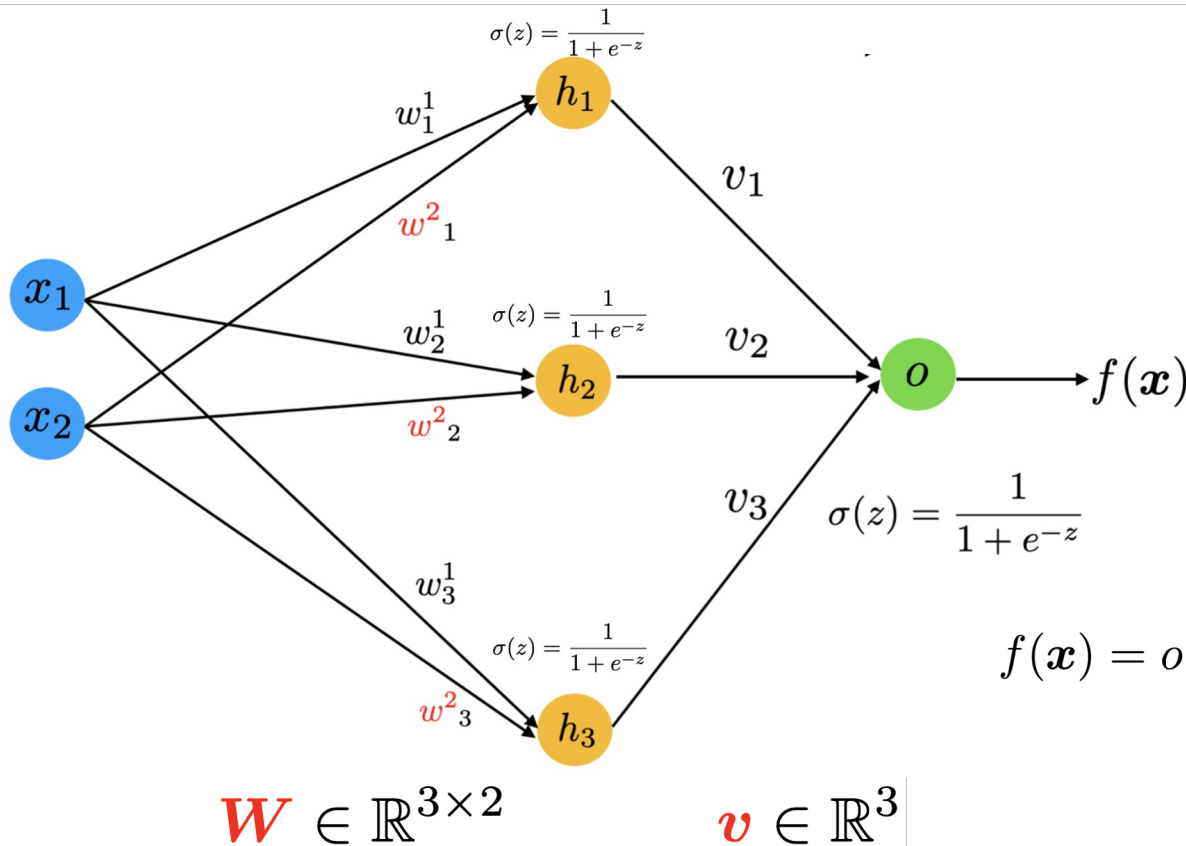
$$h_3 = \sigma(w_3^1 x_1 + w_3^2 x_2)$$

$$o = \sigma(v_1 h_1 + v_2 h_2 + v_3 h_3)$$

$$f(\mathbf{x}) = o = \sigma(v_1 h_1 + v_2 h_2 + v_3 h_3)$$

$$= \sigma\left(\mathbf{v}^\top \sigma\left(\mathbf{W}^\top \mathbf{x}\right)\right)$$

Forward propagation - Compact Form



$$h_1 = \sigma(w_1^1 x_1 + w_2^1 x_2)$$

$$h_2 = \sigma(w_2^1 x_1 + w_2^2 x_2)$$

$$h_3 = \sigma(w_3^1 x_1 + w_3^2 x_2)$$

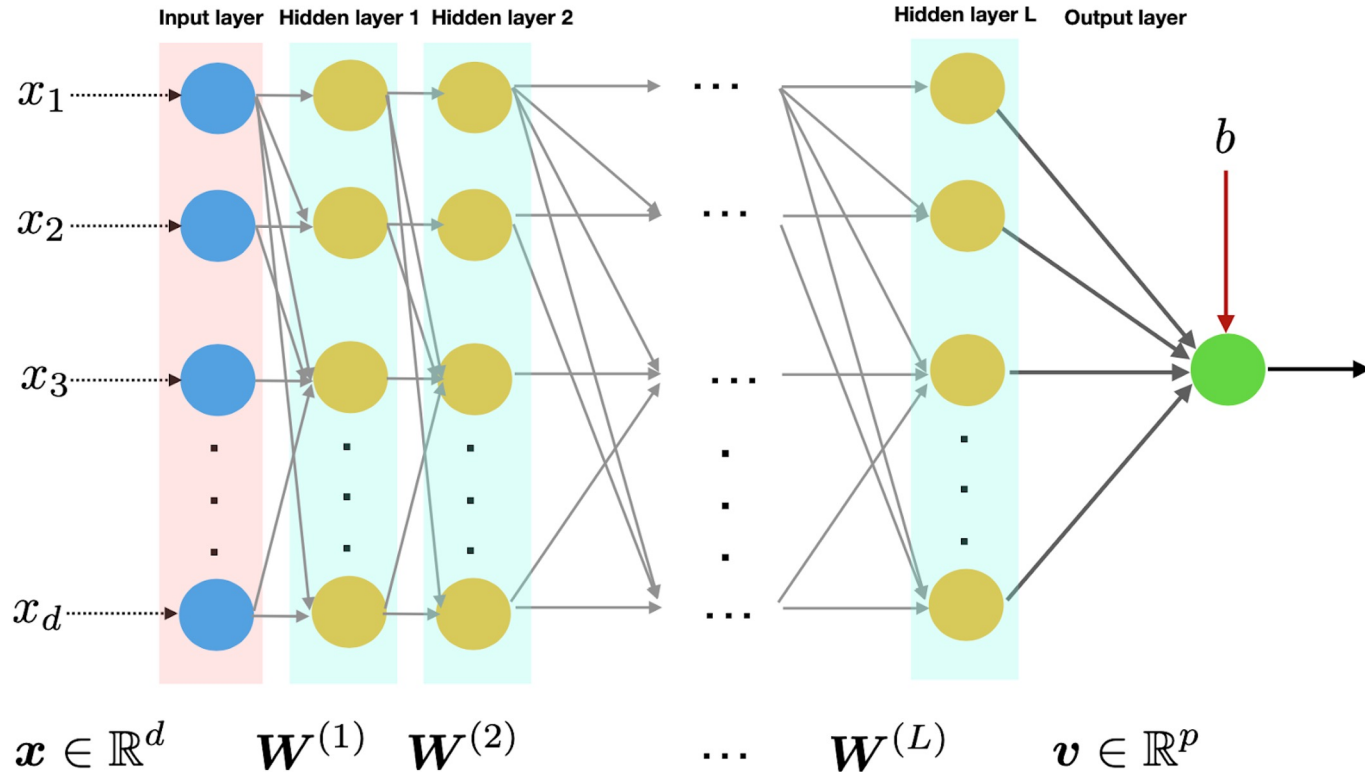
$$o = \sigma(v_1 h_1 + v_2 h_2 + v_3 h_3)$$

$$f(\mathbf{x}) = o = \sigma(v_1 h_1 + v_2 h_2 + v_3 h_3)$$

$$= \sigma(\mathbf{v}^\top \sigma(\mathbf{W}^\top \mathbf{x}))$$

Beyond two layers: multilayer neural networks

Why not add more and more layers to learn more complex decision boundaries?



Discriminative Model (e.g. Neural Network)

The discriminative model is parameterized by θ

$$P(y|X; \theta)$$

Discriminative Model Objective Function

The discriminative model is parameterized by θ

$$P(y|X; \theta)$$

We often use **negative log likelihood** over training data as our **objective function** or **loss function**. It is a function of θ

$$\mathcal{L}(\theta) = - \sum_{(X,y) \in \mathcal{D}_{\text{train}}} \log P(y|X; \theta)$$

Discriminative Model Objective Function

The discriminative model is parameterized by θ

$$P(y|X; \theta)$$

We often use **negative log likelihood** over training data as our **objective function** or **loss function**. It is a function of θ

$$\mathcal{L}(\theta) = - \sum_{(X,y) \in \mathcal{D}_{\text{train}}} \log P(y|X; \theta)$$

We then optimize the parameters to minimize the loss. Better model has lower loss

$$\hat{\theta} = \arg \min_{\theta} \mathcal{L}(\theta)$$

Optimize Objective Function by Gradient Descent

Calculate the gradient of the loss function with respect to the parameter

$$\frac{\partial \mathcal{L}(\theta)}{\partial \theta}$$

Update θ by moving a small step in the gradient direction to decrease the loss

$$\theta_{\text{new}} \leftarrow \theta_{\text{old}} - \eta \frac{\partial \mathcal{L}(\theta)}{\partial \theta}$$

η is the **learning rate**

Gradient Descent

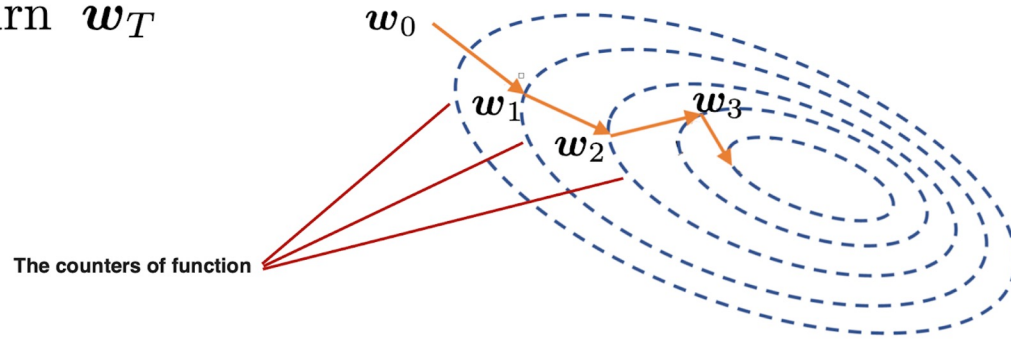
$$\mathbf{w}_0 = \mathbf{0}$$

for $t = 1, 2, \dots, T$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla f(\mathbf{w}_t)$$

end for

return \mathbf{w}_T



Gradient Descent

The key operation in gradient descent is to calculate the gradient.

$$\frac{\partial \mathcal{L}(\theta)}{\partial \theta}$$

Gradient calculation recap

Gradient of univariate scalar-valued functions

We are all familiar with basic calculus, and functions of the form $f : \mathbb{R} \rightarrow \mathbb{R}$ and their derivatives:

$$f'(a) = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h}.$$

The derivative of a function of a real variable measures the sensitivity to change of the function value (output value) with respect to a change in its argument (input value).

- ▶ $f(x) = x^r$, then $f'(x) = rx^{r-1}$,
- ▶ $\frac{d}{dx}e^x = e^x$.
- ▶ Sums rule: $(\alpha f + \beta g)' = \alpha f' + \beta g'$ for all functions f and g and all real numbers α and β
- ▶ Product rule: $(fg)' = f'g + fg'$ for all functions f and g .
- ▶ Chain rule: If $f(x) = h(g(x))$, then

$$f'(x) = h'(g(x)) \cdot g'(x).$$

Gradient of **multivariate** scalar-valued functions

Definition

Let $f : \mathcal{C} \subseteq \mathbb{R}^d \rightarrow \mathbb{R}$ be a differentiable function. Then, the gradient of f at $\mathbf{x} \in \mathcal{C}$ is the vector in \mathbb{R}^d denoted by $\nabla f(\mathbf{x})$ and defined by

$$f(\mathbf{x}) = f(x_1, x_2, \dots, x_n) \quad \nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \vdots \\ \frac{\partial f}{\partial x_d}(\mathbf{x}) \end{bmatrix}$$

Gradient of multivariate scalar-valued functions

Definition

Let $f : \mathcal{C} \subseteq \mathbb{R}^d \rightarrow \mathbb{R}$ be a differentiable function. Then, the gradient of f at $\mathbf{x} \in \mathcal{C}$ is the vector in \mathbb{R}^d denoted by $\nabla f(\mathbf{x})$ and defined by

$$f(\mathbf{x}) = f(x_1, x_2, \dots, x_n) \quad \nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \vdots \\ \frac{\partial f}{\partial x_d}(\mathbf{x}) \end{bmatrix}$$

What about vector-valued functions?

Gradient of multivariate **vector-valued** functions: Jacobian Matrix

Given a function with **m outputs** and n inputs

$$\mathbf{f}(\mathbf{x}) = [f_1(x_1, x_2, \dots, x_n), \dots, f_m(x_1, x_2, \dots, x_n)]$$

It's Jacobian is an **$m \times n$ matrix** of partial derivatives

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

$$\left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}} \right)_{ij} = \frac{\partial f_i}{\partial x_j}$$

Example Jacobian: Elementwise activation Function

$$\mathbf{h} = f(\mathbf{z}), \text{ what is } \frac{\partial \mathbf{h}}{\partial \mathbf{z}}? \quad \mathbf{h}, \mathbf{z} \in \mathbb{R}^n$$
$$h_i = f(z_i)$$

$$\left(\frac{\partial \mathbf{h}}{\partial \mathbf{z}} \right)_{ij} = \frac{\partial h_i}{\partial z_j} = \frac{\partial}{\partial z_j} f(z_i) \quad \text{definition of Jacobian}$$
$$= \begin{cases} f'(z_i) & \text{if } i = j \\ 0 & \text{if otherwise} \end{cases} \quad \text{regular 1-variable derivative}$$

$$\frac{\partial \mathbf{h}}{\partial \mathbf{z}} = \begin{pmatrix} f'(z_1) & & 0 \\ & \ddots & \\ 0 & & f'(z_n) \end{pmatrix} = \text{diag}(\mathbf{f}'(\mathbf{z}))$$

Other Jacobians

$$\frac{\partial}{\partial \mathbf{x}} (\mathbf{W}\mathbf{x} + \mathbf{b}) = \mathbf{W}$$

$$\frac{\partial}{\partial \mathbf{b}} (\mathbf{W}\mathbf{x} + \mathbf{b}) = \mathbf{I} \text{ (Identity matrix)}$$

$$\frac{\partial}{\partial \mathbf{u}} (\mathbf{u}^T \mathbf{h}) = \mathbf{h}^T$$

Left as exercise. Will be used later.

Chain Rule

Chain Rule tells us how to calculate gradients of composite functions.

For composition of one-variable functions: **multiply derivatives**

$$z = 3y$$

$$y = x^2$$

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx} = (3)(2x) = 6x$$

For multiple variables at once: **multiply Jacobians**

$$\mathbf{h} = f(\mathbf{z})$$

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

$$\frac{\partial \mathbf{h}}{\partial \mathbf{x}} = \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \dots$$

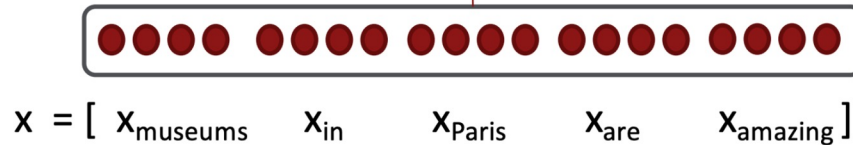
Example

Calculate $\frac{\partial s}{\partial b}$ for the following feed-forward neural network.

$$s = \mathbf{u}^T \mathbf{h}$$

$$\mathbf{h} = f(\mathbf{W}\mathbf{x} + \mathbf{b})$$

\mathbf{x} (input)



Example

Calculate $\frac{\partial s}{\partial \mathbf{b}}$

$$s = \mathbf{u}^T \mathbf{h}$$

$$\mathbf{h} = f(\mathbf{z})$$

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

\mathbf{x} (input)

$$\frac{\partial s}{\partial \mathbf{b}} = \frac{\partial s}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{b}}$$

Example

Calculate $\frac{\partial s}{\partial \mathbf{b}}$

$$\frac{\partial}{\partial \mathbf{x}}(\mathbf{W}\mathbf{x} + \mathbf{b}) = \mathbf{W}$$

$$\frac{\partial}{\partial \mathbf{b}}(\mathbf{W}\mathbf{x} + \mathbf{b}) = \mathbf{I} \text{ (Identity matrix)}$$

$$\frac{\partial}{\partial \mathbf{u}}(\mathbf{u}^T \mathbf{h}) = \mathbf{h}^T$$

$$s = \mathbf{u}^T \mathbf{h}$$

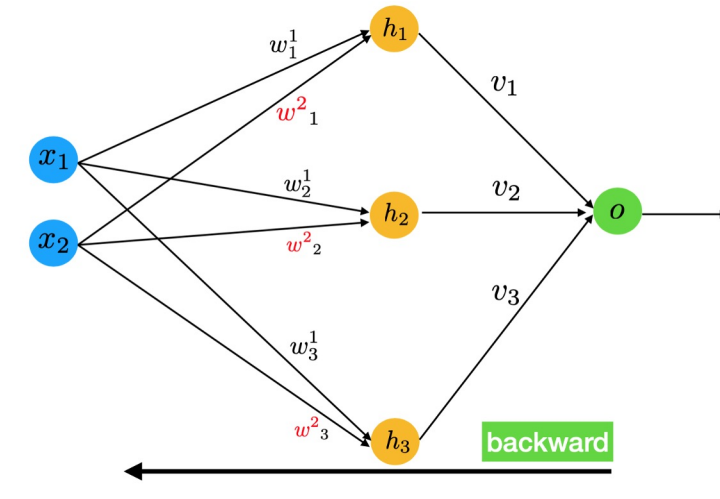
$$\mathbf{h} = f(\mathbf{z})$$

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

\mathbf{x} (input)

$$\begin{aligned} \frac{\partial s}{\partial \mathbf{b}} &= \frac{\partial s}{\partial \mathbf{h}} \quad \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \quad \frac{\partial \mathbf{z}}{\partial \mathbf{b}} \\ &\quad \downarrow \quad \quad \downarrow \quad \quad \downarrow \\ &= \mathbf{u}^T \text{diag}(f'(\mathbf{z})) \mathbf{I} \\ &= \mathbf{u}^T \circ f'(\mathbf{z}) \end{aligned}$$

Backprop



Backpropagation

We use **gradient descent** to optimize the parameters in neural networks.

The key question: How can we efficiently calculate the gradients in neural networks (or any computational graph)?

The key intuition: View neural networks (or any computational graph) as compositions of functions and use **chain rules** to calculate the gradient.

This is **Backpropagation = Gradient Descent + Chain Rule**

Computation Graph (for implementation)

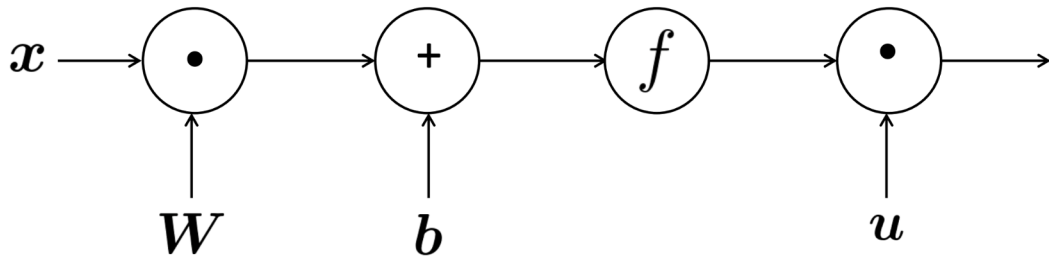
In software library (e.g. pytorch, TF), neural networks are implemented as Computation Graph (CG).

$$s = \mathbf{u}^T \mathbf{h}$$

$$\mathbf{h} = f(\mathbf{z})$$

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

\mathbf{x} (input)



Forward Propagation in CG

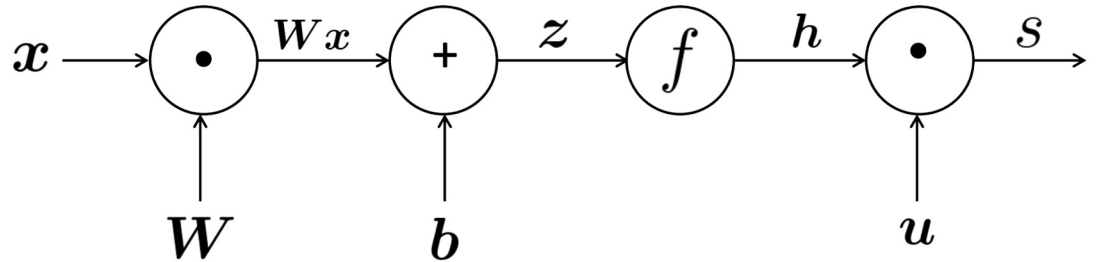
In software library (e.g. pytorch, TF), neural networks are implemented as Computation Graph (CG).

$$s = \mathbf{u}^T \mathbf{h}$$

$$\mathbf{h} = f(\mathbf{z})$$

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

\mathbf{x} (input)



Backpropagation in CG

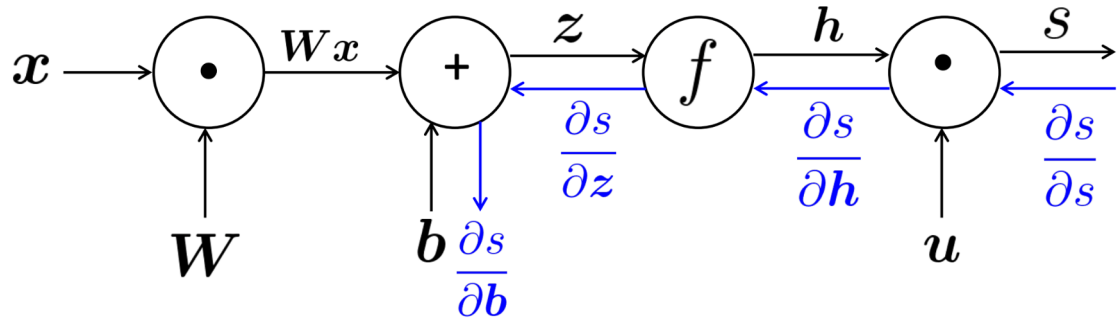
In software library, neural networks are implemented as Computation Graph (CG).

$$s = \mathbf{u}^T \mathbf{h}$$

$$\mathbf{h} = f(\mathbf{z})$$

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

\mathbf{x} (input)



Backpropagation avoids duplicated computation

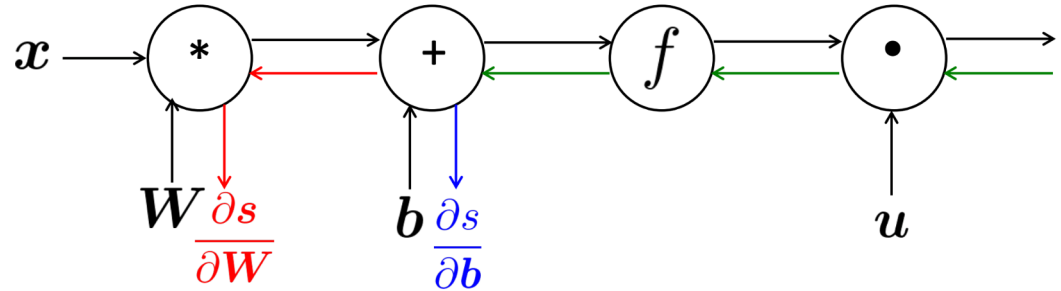
Backpropagation follows the computation graph in reverse order, so you avoid duplicated computation.

$$s = \mathbf{u}^T \mathbf{h}$$

$$\mathbf{h} = f(\mathbf{z})$$

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

\mathbf{x} (input)



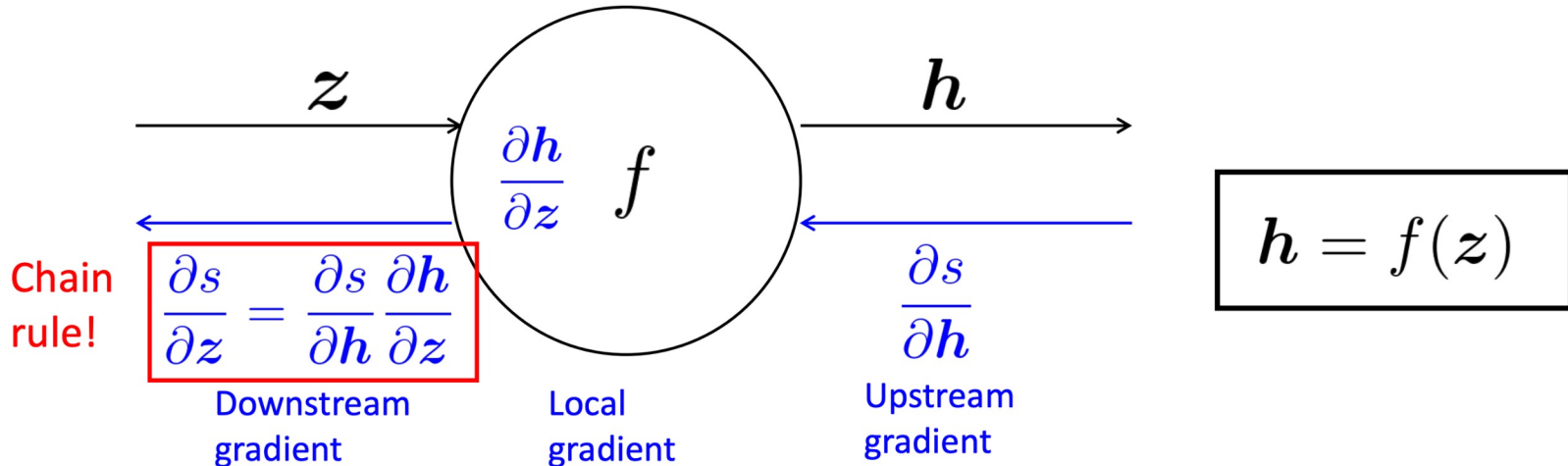
Backpropagation: Single Node

Node receives an **upstream gradient**.

Each node has a **local gradient**.

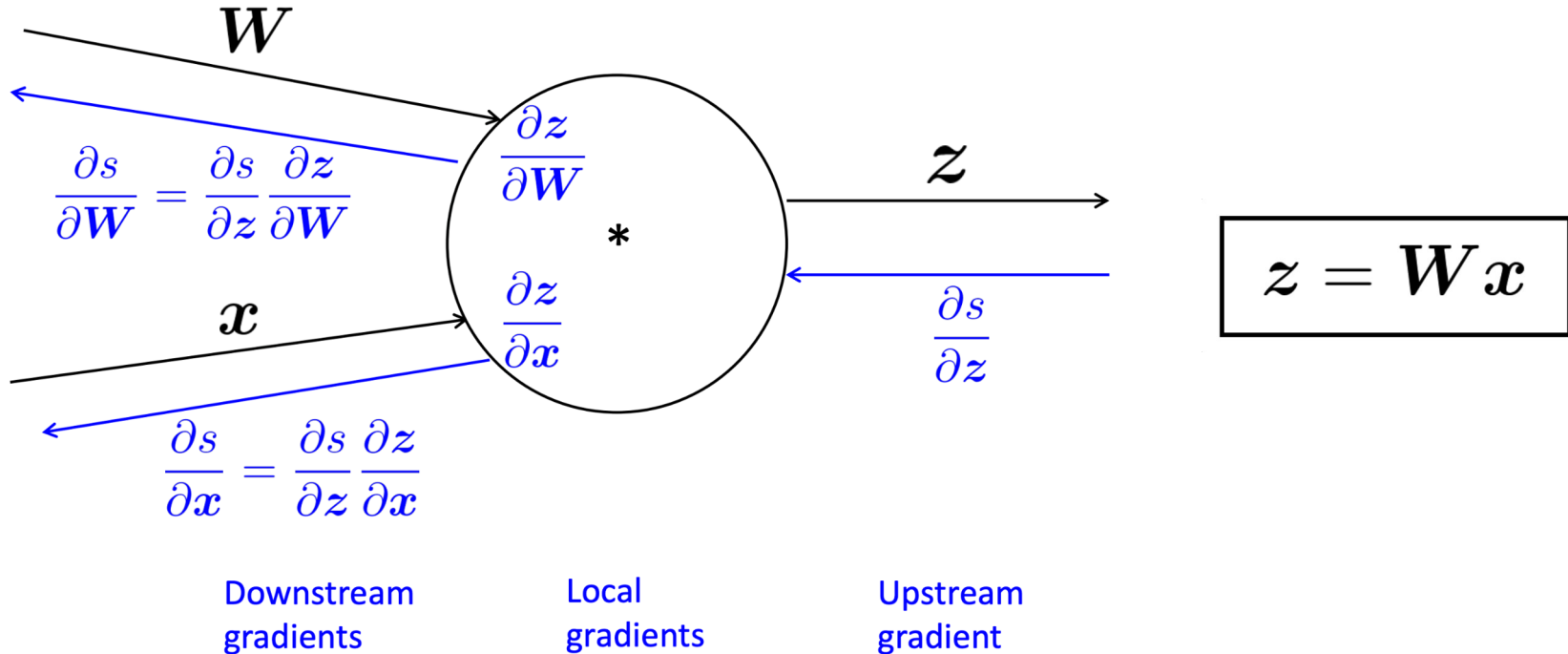
Goal is to pass on the correct **downstream gradient**.

By Chain Rule: **downstream gradient = upstream gradient x local gradient**.



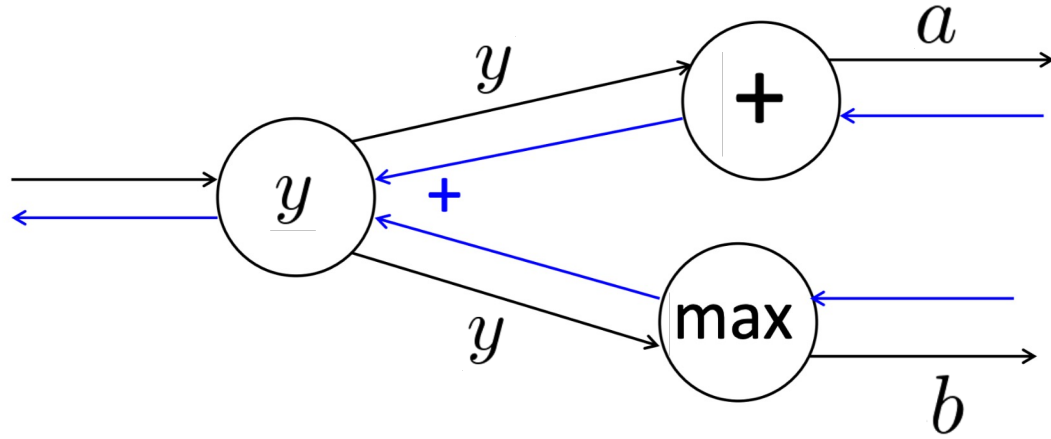
Backpropagation: Multiple Inputs

each input



Backpropagation: Multiple Outputs

If a node has multiple outgoing edges



$$a = x + y$$

$$b = \max(y, z)$$

$$f = ab$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a} \frac{\partial a}{\partial y} + \frac{\partial f}{\partial b} \frac{\partial b}{\partial y}$$

Example (Skip)

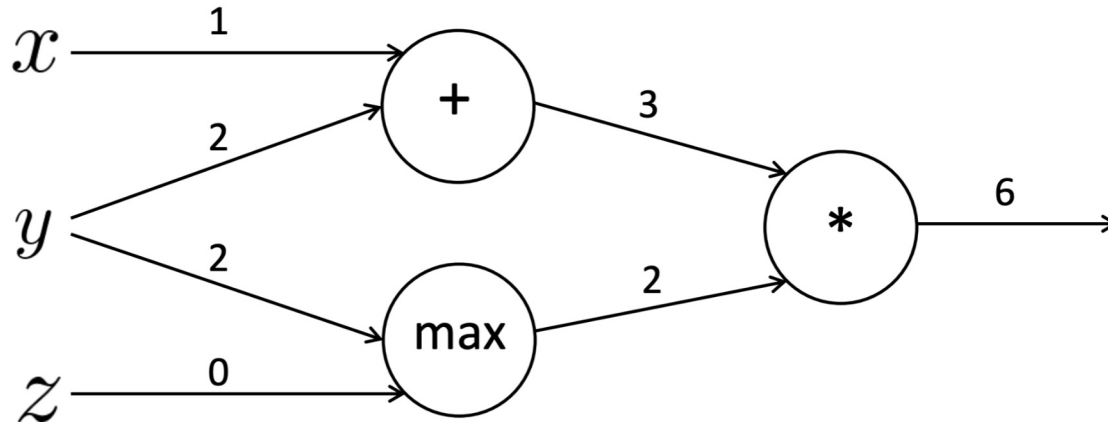
$$f(x, y, z) = (x + y) \max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$

$$b = \max(y, z)$$

$$f = ab$$



Optimizers

Once we calculate the gradient with respect to the objective function, we optimize objective function by gradient descent.

The very basic one is standard gradient descent. But there are many other choices of optimizers:

- Batch Gradient Descent
- Stochastic Gradient Descent (SGD)
- Mini-batch Gradient Descent
- SGD with Momentum
- Nesterov accelerated gradient
- Adagrad
- Adadelata
- RMSProp
- Adam; AdamW (most commonly used)
- Adafactor

Optimizers

Once we calculate the gradient with respect to the objective function, we optimize objective function by gradient descent.

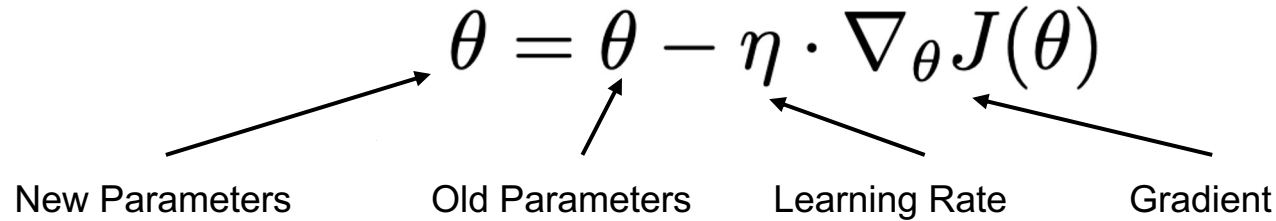
The very basic one is standard gradient descent. But there are many other choices of optimizers:

- Batch Gradient Descent
- Stochastic Gradient Descent (SGD)
- Mini-batch Gradient Descent
- SGD with Momentum
- Nesterov accelerated gradient
- Adagrad
- Adadelata
- RMSProp
- **Adam; AdamW (most commonly used)**
- Adafactor

Batch Gradient Descent

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta)$$

New Parameters Old Parameters Learning Rate Gradient

The diagram shows the equation $\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta)$ with four arrows pointing from labels below to terms in the equation. The first arrow points from 'New Parameters' to the left θ . The second arrow points from 'Old Parameters' to the right θ . The third arrow points from 'Learning Rate' to η . The fourth arrow points from 'Gradient' to $\nabla_{\theta} J(\theta)$.

```
for i in range(nb_epochs):  
    params_grad = evaluate_gradient(loss_function, data, params)  
    params = params - learning_rate * params_grad
```

Mini-batch Gradient Descent

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$



Gradient for a batch of training examples; batch size n

Stochastic Gradient Descent (SGD)

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)})$$

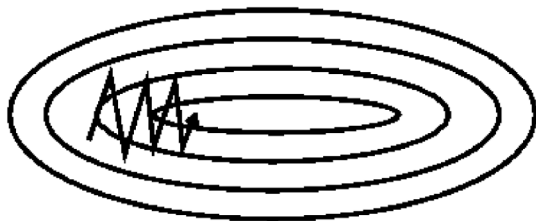


Gradient for each training example

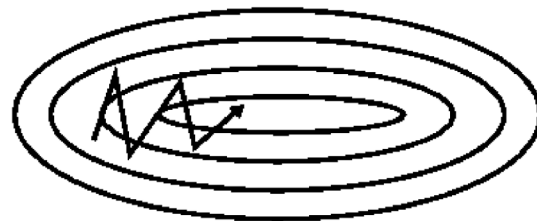
SGD with Momentum

Momentum parameter, usually 0.9 or a similar value

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta)$$
$$\theta = \theta - v_t$$



(a) SGD without momentum



(b) SGD with momentum

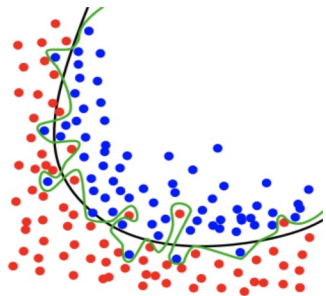
Figure 2: Source: Genevieve B. Orr

Training Tricks

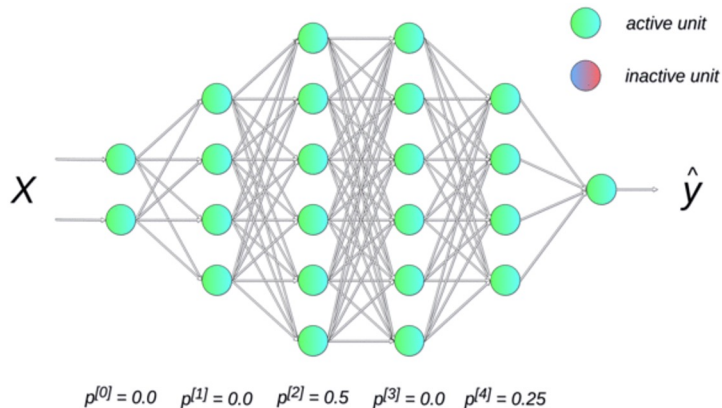
Shuffling the Training Data Every Epoch: avoid providing the training examples in a meaningful order to our model as this may bias the optimization algorithm.

Regularization: L2 regularization adds L2 norm of parameters to the loss function.

Dropout: randomly zero-out nodes in the hidden layer with probability p at training time only



While the black line fits the data well,
the green line is overfit.



Training Tricks

Shuffling the Training Data Every Epoch: avoid providing the training examples in a meaningful order to our model as this may bias the optimization algorithm.

Regularization: L2 regularization adds L2 norm of parameters to the loss function.

Dropout: randomly zero-out nodes in the hidden layer with probability p at training time only

Learning Rate Decay: gradually reduce learning rate as training continues

Training Tricks

Shuffling the Training Data Every Epoch: avoid providing the training examples in a meaningful order to our model as this may bias the optimization

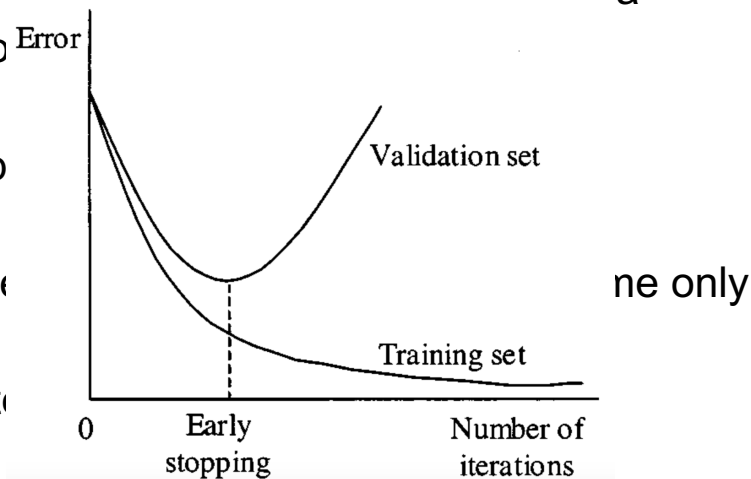
Regularization: L2 regularization adds L2 norm of parameters to the loss

Dropout: randomly zero-out nodes in the hidden layers

Learning Rate Decay: gradually reduce learning rate

Early stopping

- You should thus always monitor error on a validation set during training and stop (with some patience) if your validation error does not improve enough.
- Use **Patience**



Parameter Initialization

Very Important for neural networks to achieve good performance.

Uniform Initialization: Initialize weights in some range, such as $[-0.1, 0.1]$ for example

Xavier Initialization: $n(l)$ is the number of input units to W (fan-in) and $n(l+1)$ is the number of output units from W (fan-out).

$$W \sim U \left[-\sqrt{\frac{6}{n^{(l)} + n^{(l+1)}}}, \sqrt{\frac{6}{n^{(l)} + n^{(l+1)}}} \right]$$